Lecture 4

Alignment by Dynamic Programming

January 13, 2000
Notes: Martin Tompa

4.1. Computing an Optimal Alignment by Dynamic Programming

Given strings S and T, with $|S| = n$ and $|T| = m$, our goal is to compute an optimal alignment of S and T. Toward this goal, define $V(i, j)$ as the value of an optimal alignment of the strings $S[1] \cdots S[i]$ and $T[1] \cdots T[j]$.

The value of an optimal alignment of S and T is then $V(n, m)$. The crux of dynamic programming is to solve the more general problems of computing *all* values $V(i, j)$ with $0 \leq i \leq n$ and $0 \leq j \leq m$, in order of increasing i and j. Each of these will be relatively simple to compute, given the values already computed for smaller i and/or j, using a "recurrence relation". To start the process, we need a "basis" for $i = 0$ and/or $j = 0$.

BASIS :

$$\begin{aligned} V(0,0) &= 0 \\ V(i,0) &= V(i-1,0) + \sigma(S[i], -), \text{ for } i > 0 \\ V(0,j) &= V(0,j-1) + \sigma(-, T[j]), \text{ for } j > 0 \end{aligned}$$

The basis for $V(i, 0)$ says that if i characters of S are to be aligned with 0 characters of T, then they must all be matched with spaces. The basis for $V(0, j)$ is analogous.

RECURRENCE : For $i > 0$ and $j > 0$,

$$V(i,j) = \max(\begin{array}{lcll} V(i-1,j-1) & + & \sigma(S[i],T[j]) & , \\ V(i-1,j) & + & \sigma(S[i],-) & , \\ V(i,j-1) & + & \sigma(-,T[j]) &) \end{array}$$

This formula can be understood by considering an optimal alignment of the first i characters from S and the first j characters from T. In particular, consider the last aligned pair of characters in such an alignment. This last pair must be one of the following:

1. $(S[i], T[j])$, in which case the remaining alignment excluding this pair must be an optimal alignment of $S[1] \cdots S[i-1]$ and $T[1] \cdots T[j-1]$ (i.e., must have value $V(i-1, j-1)$), or
2. $(S[i], -)$, in which case the remaining alignment excluding this pair must have value $V(i-1, j)$, or

3. $(-, T[j])$, in which case the remaining alignment excluding this pair must must have value $V(i, j-1)$.

The optimal alignment chooses whichever among these three possibilities has the greatest value.

4.1.1. Example

In aligning `acbcdb` and `cadbd`, the dynamic programming algorithm fills in the following values for $V(i,j)$ from top to bottom and left to right, simply applying the basis and recurrence formulas. (As in the example of Section 3.3, assume that matches score $+2$, and mismatches and spaces score -1.) For instance, in the table below, the entry in row 4 and column 1 is obtained by computing $\max(-3+2, 0-1, -4-1) = -1$.

i \ j		0	1	2	3	4	5
			c	a	d	b	d
0		0	−1	−2	−3	−4	−5
1	a	−1	−1	1	0	−1	−2
2	c	−2	1	0	0	−1	−2
3	b	−3	0	0	−1	2	1
4	c	−4	−1	−1	−1	1	1
5	d	−5	−2	−2	1	0	3
6	b	−6	−3	−3	0	3	2

The value of the optimal alignment is $V(n,m)$, and so can be read from the entry in the last row and last column. Thus, there is an alignment of `acbcdb` and `cadbd` that has value 2, so the alignment proposed in Section 3.3 with value 1 is not optimal. But how can one determine the optimal alignment itself, and not just its value?

4.1.2. Recovering the Alignments

The solution is to retrace the dynamic programming steps back from the (n,m) entry, determining which preceding entries were responsible for the current one. For instance, in the table below, the (4,2) entry could have followed from either the (3,1) or (3,2) entry; this is denoted by the two arrows pointing to those entries. We can then follow any of these paths from (n,m) to $(0,0)$, tracing out an optimal alignment:

i \ j		0	1	2	3	4	5
			c	a	d	b	d
0		0	← −1	−2	−3	−4	−5
1	a	↑ −1	−1	↖ 1	0	−1	−2
2	c	−2	↖ 1	0	↖ 0	−1	−2
3	b	−3	↑ 0	↖ 0	−1	↖ 2	1
4	c	−4	−1	↖↑ −1	−1	↑ 1	1
5	d	−5	−2	−2	↖ 1	0	↖ 3
6	b	−6	−3	−3	0	↖ 3	←↑ 2

The optimal alignments corresponding to these three paths are

$\boxed{\begin{array}{ccccccc} \text{a} & \text{c} & \text{b} & \text{c} & \text{d} & \text{b} & \text{-} \\ \text{-} & \text{c} & \text{-} & \text{a} & \text{d} & \text{b} & \text{d} \end{array}}$, $\boxed{\begin{array}{ccccccc} \text{a} & \text{c} & \text{b} & \text{c} & \text{d} & \text{b} & \text{-} \\ \text{-} & \text{c} & \text{a} & \text{-} & \text{d} & \text{b} & \text{d} \end{array}}$, and $\boxed{\begin{array}{ccccccc} \text{-} & \text{a} & \text{c} & \text{b} & \text{c} & \text{d} & \text{b} \\ \text{c} & \text{a} & \text{d} & \text{b} & \text{-} & \text{d} & \text{-} \end{array}}$.

Each of these has three matches, one mismatch, and three spaces, for a value of $3 \cdot (2) + 4 \cdot (-1) = 2$, the optimal alignment value.

4.1.3. Time Analysis

Theorem 4.1: The dynamic programming algorithm computes an optimal alignment in time $O(nm)$.

Proof: This algorithm requires an $(n+1) \times (m+1)$ table to be completed. Any particular entry is computed with a maximum of 6 table lookups, 3 additions, and a three-way maximum, that is, in time c, a constant. Thus, the complexity of the algorithm is at most $c(n+1)(m+1) = O(nm)$. Reconstructing a single alignment can then be done in time $O(n+m)$. □

4.2. Searching for Local Similarity

Next we will discuss some variants of the dynamic programming approach to string alignment. We do this to demonstrate the versatility of the approach, and because the variants themselves arise in biological applications.

In the variant called "local similarity", we are searching for regions of similarity between two strings, within contexts that may be dissimilar. An example in which this arises is if we have two long DNA sequences that each contain a given gene, or perhaps closely related genes. Certainly the "global" alignment problem of Definitions 3.3 and 3.4 will not in general identify these genes.

We can formulate this problem as the *local alignment problem*: Given two strings S and T, with $|S| = n$ and $|T| = m$, find substrings (i.e., contiguous subsequences) A of S and B of T such that the optimal (global) alignment of A and B has the maximum value over all such substrings A and B. In other words, the optimal alignment of A and B must have at least as great a value as the optimal alignment of any other substrings A' of S and B' of T.

4.2.1. An Obvious Local Alignment Algorithm

The definition above immediately suggests an algorithm for local alignment:

```
for all substrings $A$ of $S$ do
    for all substrings $B$ of $T$ do
        Find an optimal alignment of $A$ and $B$ by dynamic programming;
        Retain $A$ and $B$ with maximum alignment value, and their alignment;
    end ;
end ;
Output the retained $A$, $B$, and alignment;
```

There are $\binom{n+1}{2}$ choices of A, and $\binom{m+1}{2}$ choices of B (excluding the length 0 substrings as choices). Using Theorem 4.1, it is not difficult to show that the time taken by this algorithm is $O(n^3m^3)$. We will see in Section 5.1, however, that it is possible to compute the optimal local alignment in time $O(nm)$, that is, the same time used for the optimal global alignment.

4.2.2. Set-Up for Local Alignment by Dynamic Programming

Definition 4.2: The *empty string* λ is the string with $|\lambda| = 0$.

Definition 4.3: U is a *prefix* of S if and only if $U = S[1] \cdots S[k]$ or $U = \lambda$, for some $1 \leq k \leq n$, where $n = |S|$.

Definition 4.4: U is a *suffix* of S if and only if $U = S[k] \cdots S[n]$ or $U = \lambda$, for some $1 \leq k \leq n$, where $n = |S|$.

For example, let $S =$ `abcxdex`. The prefixes of S include `ab`. The suffixes of S include `xdex`. The empty string λ is both a prefix and a suffix of S.

Definition 4.5: Let S and T be strings with $|S| = n$ and $|T| = m$. For $0 \leq i \leq n$ and $0 \leq j \leq m$, let $v(i, j)$ be the maximum value of an optimal (global) alignment of α and β over all suffixes α of $S[1] \cdots S[i]$ and all suffixes β of $T[1] \cdots T[j]$.

For example, suppose $S =$ `abcxdex` and $T =$ `xxxcde`. Score a match as $+2$ and a mismatch or space as -1. Then $v(5, 5) = 3$, with $\alpha =$ `cxd`, $\beta =$ `cd`, and alignment

c	x	d
c	-	d
+2	–1	+2

The dynamic programming algorithm for optimal local alignment is similar to the dynamic programming algorithm for optimal global alignment given in Section 4.1. It proceeds by filling in a table with the values of $v(i, j)$, with i, j increasing. The value of each entry is calculated according to a new basis and recurrence for $v(i, j)$, given in Section 5.1. Unlike the global alignment algorithm, however, the value of the optimal local alignment can be any entry, whichever contains the maximum of all $(n + 1)(m + 1)$ values of $v(i, j)$. The reason for this is that each $v(i, j)$ entry represents an optimal pair (α, β) of suffixes of a given pair $(S[1] \cdots S[i], T[1] \cdots T[j])$ of prefixes. Since a suffix of a prefix is just a substring, we find the optimal pair of substrings by maximizing $v(i, j)$ over all possible pairs (i, j).